PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : H04L 12/00 — A2

(11) International Publication Number: WO 99/14891

(43) International Publication Date: 25 March 1999 (25.03.99)

(21) International Application Number: PCT/GB98/02721

(22) International Filing Date: 14 September 1998 (14.09.98)

(30) Priority Data:
9719412.0 12 September 1997 (12.09.97) GB
9719403.9 12 September 1997 (12.09.97) GB

(71) Applicant (*for all designated States except US*): COMMUNICATION & CONTROL ELECTRONICS LIMITED [GB/GB]; 2 Occam Court, Occam Road, The Surrey Research Park, Guildford, Surrey GU2 5YQ (GB).

(72) Inventor; and

(75) Inventor/Applicant (*for US only*): SCHMIDT, Markus, Johannes [DE/GB]; 15 Grove Road, Surbition, Surrey KT6 4BX (GB).

(74) Agent: FITZPATRICKS; 4 West Regent Street, Glasgow G2 1RS (GB).

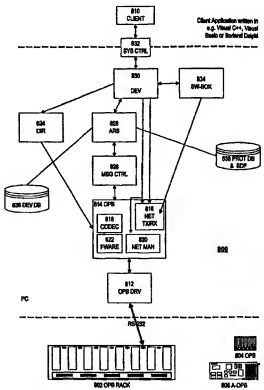
(81) Designated States: DE, DE (Utility model), GB, JP, US, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: DEVELOPMENT AND TEST TOOLS FOR COMMUNICATION SYSTEM

(57) Abstract

The development system comprises reuseable computer implemented components to be used in conjunction with a hardware network interface board, for emulation of devices in a functioning network. The same components can provide tools for stimulation and analysis of network traffic and compliance verification. The components provide an interface between a client program and the network interface hardware. They include a protocol database, in which application–level protocols specific to devices such as CD changers, amplifiers and audio–video control units are defined. The proposed tools offer a mechanism for filtering commands instructed by the user (client program), where these conflict with application protocols defined for the specific type of device being emulated. The components include program modules for automatic implementation of standard behaviours, such as network management and source data connection routing. In addition to application protocols generic to all network configurations employing a certain protocol, the tools may store a network system description defining the particular number and type of devices present or emulated on the network, to restrit further the range of legal commands.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

DEVELOPMENT & TEST TOOLS FOR COMMUNICATION SYSTEM

The present invention relates to development tools implemented in hardware and software for use in developing and testing apparatuses for use in a local communication system. The invention relates particularly to the development of products implementing not only low level communication protocols, but also higher level application protocols, for example in consumer audio/video networks.

A local communication system which combines source data (CD audio, MPEG video, telephone audio etc) with control messages in a low cost fibre network has been proposed in the form of D2B Optical. For details, see for example the "Conan Technology Brochure" and the "Conan IC Data Sheet" available from Communication & Control Electronics Limited, 2 Occam Court, Occam Road, The Surrey Research Park, Guildford, Surrey, GU2 5YQ (also http://www.candc.co.uk). See also European patent applications of Becker GmbH EP-A-0725516 (95P03), EP-A-0725518 (95P04), EP-A-07225515 (95P05), EP-A-0725520 (95P06), EP-A-0725521 (95P07), EP-A-0725522 (95P08), EP-A-0725517 (95P09) and EP-A-0725519 (95P10). "Conan" is a registered trade mark of Communication & Control Electronics Limited. The present description further refers to double speed interfaces, which are described in our co-pending application PCT/GB98/00349 (62793WO).

Tools, usually involving a combination of software and interface hardware are generally known for allowing a personal computer (PC) or other device to emulate products being developed for use in such networks, and to verify/test real products which are already developed. The present invention aims to improve the utility of such development tools. Applications include the simulation of new products, including user interface and high level control features, prior to manufacture and the verification/compatibility test of real products. It is in the nature of the systems being developed that a number of units of different types may be being developed simultaneously, and the definition and debugging of their functional requirements becomes very complex. In particular, faults in one device in the network may manifest themselves by undesirable behaviour in another device, leading to difficulties in diagnosis.

The present invention in a first aspect, provides a system as defined in the appended claim 1. In another aspect, the invention provides a tool comprising computer implemented components to be used in conjunction with a hardware network interface, said tool components

providing an interface between a user or a client program and the network interface, the tool incorporating an at least one Application Programmers Interface (API), a protocol database and/or a system model, and components for processing messages and events from the user or client program by reference to the database/model, so as to detect illegal messages being transmitted onto or received from the network.

In particular, application-level protocols specific to devices such as CD changers, amplifiers and audio/video control units are defined for the D2B Optical system. The proposed tools offer a mechanism for filtering commands instructed by the user (client program), where these conflict with protocols defined for the specific type of device being emulated. In general, application protocols are those which relate to substantial functionality of a device connected to the network, rather than to mere communication management, or network management tasks.

While, in general, the protocols may define individual message and response formats, which are employed as required by a specific product designer, to achieve his or her own design behaviour, there may also be areas in which the protocols define sequences of behaviour, particularly where extended co-operation between plural stations in the network is necessary to achieve a design result. Examples are in network management (initialisation) and, in D2B Optical at least, source data connection management. The tool may include program modules for automatic implementation of these standard behaviours, such that the user of the tools need not explicitly code the standard parts of his application.

In the embodiments presented herein, program modules comprising the tool take the form of objects, which exchange messages with one another according to predetermined rules. Certain objects may be shared between different applications active in the same computer hardware, while others are reserved to a single active application. The development system may thus comprise several instances of each reserved object class, serving different client applications. Those which are shared may in particular include objects which provide the interface to specific interface hardware of the tool system (hardware drivers). Those which are reserved to specific "system instances" allow plural network devices to be emulated, tested or controlled, entirely independently, with a single computer.

In addition to application protocols generic to all network configurations employing a certain protocol, the tools may store a network system description defining the particular

number and type of devices present or emulated on the network, to restrict further the range of legal commands.

The invention further provides methods of product emulation and testing, substantially as described herein with reference to the accompanying drawings. It will be appreciated that the program and hardware components of the system may be supplied together or separately, and may be supplied as components for use with an existing general purpose computer.

BRIEF DESCRIPTION OF THE DRAWINGS

Embodiments of the invention will now be described by way of example only with reference to the accompanying drawings in which:

Figure 1 shows in block schematic form a local communication system incorporating a range of different devices communicating in a ring network;

Figure 2 illustrates the control and source data architecture used in the system of Figure 1;

Figure 3 represents a station with integral interface;

Figure 4 schematically illustrates one of the interface modules of Figure 1;

Figure 5 shows the frame structure of digital signals transmitted according to the known D2B Optical format;

Figure 6 shows the frame structure of digital signals transmitted between the apparatuses in the system of Figure 1;

Figure 7 shows the structure of a control frame carried within the frame structure of Figure 6;

Figure 8 shows the software and hardware elements of a product development and test tool for use in developing apparatus compatible with the network of Figure 1;

Figure 9 illustrates use of the development tool in emulation in a product under development;

Figure 10 illustrates use of the product development tool in testing of several devices to verify compatibility with the network of Figure 1;

Figure 11 illustrates the action of various component modules in the initialisation of the development system; and

Figure 12 illustrates the action of the component modules in sending a message as part of product emulation.

DETAILED DESCRIPTION OF EXEMPLARY EMBODIMENTS

D2B Optical System Overview

The various aspects of the invention will be illustrated, by way of example only, as applied in a D2B Optical network. The nature and operation of this network will first be described briefly, as background.

The system illustrated in Figure 1 comprises nine audio- or video-related apparatuses 101-109 connected as stations (or nodes) of a Local Area Network (LAN). Of course more or fewer than nine stations may be accommodated. In this example system, the apparatuses are: a control and display unit 101, a Compact Disc memory (CD-ROM) reader 102, a radio turner 103, a CD changer unit 104, an audio power amplifier 105, a facsimile send/receiver unit (FAX) 106, a video recording system (VCR/CAMCORDER) 107, a video tuner 108, and a telephone 109. The display function of the control and display unit 101 may for example provide for display of information read from memory devices by CD-ROM and/or display of video signals from tuner 108 or VCR 107.

The LAN interconnection in the known system comprises nine unidirectional point-to-point fibre optic links 111, 112 etc. linking interface modules 121 etc., each of which is substantially structurally identical, such that the nodes are all connected in a ring. Each fibre optic link carries a combination of digital audio/video signals, CD-ROM data and control messages in accordance with a frame structure to be described in detail below. A designated station (referred to hereinafter as the system master), such as the control/display unit 101,

continuously generates the frame structure at a frame sample rate of 20-50kHz (typically 44.1kHz as for CD sampling). One station on the network is designated to act as system master on start-up although the role of system master may subsequently be re-allocated to another station, for example in fault conditions as described below.

The implementation of a station's interface to the fibre optic ring is schematically illustrated in Figure 2. From the ring 119-111, a media access control (MAC)/physical layer 300 together with a communications management layer 302 for control messages are provided in interface module 121. The communications management layer 302 manages address initialisation and verification and ensures the reliable transport of messages by retransmission according to defined timing rules. Data handling for source data 304 and application protocols for control messages 306 are provided at station level 101, with the application protocols typically defining a device/subdevice grouping and control hierarchy for the station, the format of information exchanged between products, the behaviour of devices/subdevices, and application level timing. It will be readily understood that the interface module 121 may be physically within a station, for example in the form of the Conan integrated circuit or similar network transceiver and associated control software.

As shown in Figure 3, a similar the interface module 123 is provided as one function within the radio cassette player 103 which also has amplifier 310, tuner 312, tape playback deck 314, audio/video controller (AVC) 316 and user I/O 318 functions. These functions and their interconnections are not shown and have no direct bearing on the present invention. Their implementation will be readily apparent to those of ordinary skill in the art.

Figure 4 is a schematic representation of an interface module 121 (in this case node 121) linking a station to the known fibre optic ring. All stations connected to the LAN can generate and/or receive source data (up to three channels SD0-SD2) and control data (CTRL). The control data is of low volume, arrives in burst and is user/event driven (for example user instructions or status changes), whereas the source data is a continuous high volume stream (for example audio, compressed video, CD-ROM data).

In the D2B Optical system, the source data and control messages are transported on the network from node-to-node in frames generated by a station designated as the System Master. Frames are circulated at the same rate as the audio sample frequency, typically $fs = 44.1$kHz. Frames are grouped into blocks of 48 frames.

Figure 5 shows how each network frame is divided into two sub-frames ('left' and 'right'). At $fs = 44.1$kHz, there will be 88,200 sub-frames per second. The left sub-frame is always the first of the pair transmitted on the network. At the physical level, bits are transported with bi-phase encoding. The relationship between the block, frame, sub-frame and control frame is shown in Figure 5.

Figure 6 shows how each sub-frame contains 64 bits, handled within the transceiver as 8 byte fields. The fields comprise the preamble, the transparent channels, 6 bytes of source data, and 8 control/status bits which make up the control frames and the SPDIF status bits. The meaning of the various fields will now be described in detail.

The fields of the sub-frame structure of Figure 6 are:

- Preamble: The preamble synchronises the network receiver. There are three types of preamble, identical to those defined in the IEC-958 (SPDIF) specification. They contain bi-phase coding violations which the receiver can recognise. The three unique preambles identify left, right and block sub-frames. The left preamble identifies the beginning of a frame and the block preamble identifies the beginning of a block. The block preamble replaces every 48th left preamble. This provides a block structure to which the control frame data is synchronised.
- Source Data Bytes: The source data bytes carry the high-volume real-time digital source data. The bytes may be allocated flexibly, so that the devices in a system may use the source data bytes in the most efficient way for that system (see EP-A-0725520 and EP-A-0725521).
- Control Bits: The control bits CF0 and CF1 carry the control messages (for controlling devices and sending status information). There are 2 CF bits per sub-frame, and a control frame is 192 bits long, therefore 96 sub-frames (48 left + 48 right) are required to build-up a complete control frame. The control frame is shown in Figure 7.

As shown in Figure 5, the control frame is assembled from and aligned with a block of 96 sub-frames, i.e. the first two bits of a new control frame are taken from the sub-frame with a block preamble, and subsequent pairs of bits are taken from subsequent sub-frames to build up a control frame. The fields of the control frame are:

- Arbitration bits: These indicate if the control frame is free or occupied. The transceiver handles these bits automatically.
- Destination Address: This is the 12-bit address of the destination of the message, in the range '000'H to 'FFF'H. The sending device writes this into its message transmit buffer for transmission. Certain addresses and address ranges have special meanings, such that stations can be addressed either by ring position or by an application-related 'device address'. Broadcast and 'groupcast' addressing is also provided.
- Source Address: This is the 12-bit address of the sender of the message, in the range '000'H to 'FFF'H. The receiving device can read this from its message receive buffer after reception.
- Message Type and Length: Two 4-bit fields normally used to indicate the type/length of the message. Message types include commands, status reports and requests for status reports.
- Data 0 to 15: The message data. All 16 bytes are always transported. The Message Length normally indicates how many of the 16 bytes are actually valid for the message. The sending device writes this into its message transmit buffer for transmission. The receiving device can read this from its message receive buffer after reception. The message typically comprises an operation code (op-code) and one or more operands.
- CRC: A 16-bit Cyclic Redundancy Check value used to verify that the control frame has been transported without error. The CRC is generated by the interface module automatically on message transmission and checked automatically on message reception.
- ACK/NAK: Acknowledge and Not Acknowledge (2-bits each) indicate successful message transmission. The use of separate ACK and NAK flags allow reliable point-to-point and broadcast message transport, as described in our application GB-A-2302243. The flags are automatically filled by the destination device(s) (if present) and read by the sending device.
- Reserved: 10 bits are reserved for future definition.

Product Development System Overview

The novel product development environment described herein may be used by an individual manufacturer or a system integrator, to assist in development and testing of products and/or complete network systems, which are to operate together with products of the same or other manufacturers. Of particular concern is the correct implementation of protocols for exchange of control messages and data via the Control Frame mechanism of Figure 7. The D2B Optical specification defines certain protocols from the media access layer up into the

application layer (using the terminology of the ISO/OSI network model). In other words, mechanisms for initialisation of the network and for the exchange of control frame messages (referred to below as D2B messages) are defined, but also the specific content and meaning of messages for a range of purposes.

For example, at the network management level, procedures and message formats are defined which permit each station to acquire a unique address for the receipt of D2B messages. At the application level, messages are defined for causing an amplifier connected to the network to increase its volume setting, or to switch to reproducing an audio data stream carried in certain bytes of the source data field in each network frame (Fig. 5). At the same time, maximum freedom is intended, by which manufacturers can implement their own preferred functionality in whatever kind of product is connected to the network.

Figure 8 provides an overview of the novel product development system, including in particular a software engine 800 comprising various software components running on a PC, and an interface board rack system 802 carrying one or more different types of interface boards ("Optical Power Board" (OPB) 804 and "Advanced OPB" (AOPB) 806). Each interface board includes one or more network transceiver integrated circuits, such as the Conan chip mentioned above. The development system engine 800 includes various program and database modules 812 to 836, described in more detail below. A System Control module 832 provides the Application Programming Interface (API) for a client application program 810. The client application 810 may be a standard one provided with the development system, or may be specially developed to emulate a given product under development.

The development system engine 800 in this embodiment is a general purpose server providing both low and high level D2B Optical single speed or double speed communication, and control over ICs such as those in the Conan product range.

The development system engine is in practice a combination of programmable software components using advanced up-to-date software design technology such as ActiveX. This offers maximum flexibility while retaining a high degree of functional reliability for all programs and applications accessed through well-defined application programming interfaces (API). The engine 800 is a design system which makes it extremely easy to write D2B Optical applications. It provides generic functions to manipulate transceiver ICs and to communicate

D2B Optical control messages. It can be programmed using any environment which supports ActiveX, such as Visual Basic, Visual C++ or Borland Delphi.

The engine can be used for example to emulate D2B Optical devices or for testing/verifying real D2B Optical products, and can of course be adapted to other protocols. In addition to emulation, the same components forming the development system provide a common framework for stimulation and/or analysis of network traffic, and for verification of a product's compliance with standard protocols.

Just like a real product on the network, each interface board for use in conjunction with the software components carries one or more network transceiver integrated circuits. The Advanced OPB 806 also provides a facility whereby several network interface boards can be connected through rack system 802 in a "daisy chain" arrangement, such that several devices in one network, or several separate networks, can be simulated or tested from the same PC. The architecture of the software tools will be described in some detail. Below. Two illustrations of the system in use will first be given, to aid understanding.

In the example of Figure 9, the development system software is running on a portable PC 900 connected by RS-232 serial cable 910 to an interface board 920, which features one or more network transceiver integrated circuits. This set-up can be used for example to simulate a D2B Optical telephone under development. A device manufacturer can implement the complete behaviour model of the real product in a 'C' program, for example, and, once the concept/design is proven, he can transfer the same code into his embedded microcontroller in a real product. With reference to the components in the larger network of Figure 1, products 101 (Head unit) 104 (CD changer) and 105 (amplifier) are existing, while the development system provides at an emulation 109' of the telephone product 109.

In Figure 10, the system software is running on a different PC 1000 which is connected using RS-232 link 1010 to an OPB Rack System 1002. The OPB Rack System itself hosts five OPBs which are "daisy-chained" within the rack. This system can test conformance of products containing a D2B Optical interface to a set of predetermined criteria (protocol and basic functional behaviour verification (Play, Stop etc) and the like), to ensure that the product meets agreed compatibility levels. It can be used at any stage of the product life cycle to assist system integrators or product development engineers to identify, with a high degree of accuracy, the compliance of a product with the D2B Optical or other applicable specifications,

both at low level as well at Application protocol level. Again using the reference numerals of Figure 1, five CD changer products 104-1 to 104-5 are shown under test.

The advantages of such a system are:

- Improved accuracy and repeatability of quality and validation checks because both tests and test sequences, as well as the response to tests and reporting of inconsistencies, are predetermined; and
- automatic test capabilities shortening the product development cycle as equipment can check more accurately and far quicker than a dedicated test engineer if the developed product will function correctly in a system, and to a pre-determined level of the D2B Optical or other desired protocol..
- Products can be checked based on a predetermined test cycle on the production line and/or of a sample or all of the product output.
- Test engineers can identify if previously checked devices are repaired and function to agreed standard following a 'repair' cycle.

Development System Software Structure: Introduction

In addition to the interface board, the disclosed tools comprise a system of software components that integrate and interact together to provide services to client applications, enabling those applications to operate within, use, and control a D2B Optical network, from a Microsoft Windows operating system.

The following describes the software architecture for the tool system. As the system is designed in an object oriented manner, it makes sense to discuss the general design of the objects, their purpose, and their interaction with other parts of the system as a whole. The detailed design was performed with the design tool Rational Rose, although alternative tools may be equally suitable.

Design Objectives

The design was performed with the following aims in mind:

- To produce a family of software components whose ultimate goal is to ease the development of software and/or firmware to utilise the D2B Optical and similar technology.
- To design the range components in such a way as to give varying levels of control to the user from low level control (e.g. for interacting directly with hardware), up to very high level control (e.g. automated device behaviour).
- To design the components in a layered fashion so that higher level components utilise the services of the lower level components underneath. By doing this maximum re-use of components and software is enabled.
- To make the components themselves extensible and flexible so that it is easy to create new variations of them if new requirements arise in the future.
- To abstract the end users of the components from the hardware. This enables users to write software without worrying about the underlying hardware and having to take it into account when writing that software.
- To provide simple intuitive interfaces for users to utilise the components.
- To provide a solid underlying architecture for other Emulators, Compliance Checker and user definable (proprietary) emulators.
- To provide an Automatic Response System, as explained below.

Class Design

The following sections discuss the objects in the design. The discussion is general but specific where it is required. In the terminology of object orientated architecture well known to those skilled in the art, various "classes" of "object" are defined. One or more "instances" of each class can be created to act and interact with each other as distinct "objects" in a well-defined manner. Figure 8 shows the classes in a "layered" manner, with "low level" components at the bottom and higher ones at the top. These components or "objects" can interact by the exchange of messages. The arrows describe the main flows of communication within the system.

OPB Driver 812

OPB Driver 812 object provides the lowest level interface to the actual network transceiver hardware. Each object provides an API (Application Programming Interface) to a specific interface board type, e.g. parallel, serial configuration, A-OPB etc. The OPB Driver

class encapsulates the knowledge of the low level communication and control mechanism and provides an object oriented interface to access their functionality. Further, provided that each class provides a member function for each of the exported functions in the driver it encapsulates, existing applications may be altered to use the classes in the future.

When any new interface board (e.g. with USB or PCI-Interface) is developed, a new OPB Driver class will also have to be designed, again around the driver that actually controls the new interface board.

OPB 814

The OPB classes are the classes that end user applications and development system itself use to control hardware boards in the Optical Power Board family. The functionality available to a user is therefore restricted to whatever functionality the OPB class exposes in its API. Each OPB object is connected to an OPB Driver object of the relevant type. The OPB object uses the driver object to perform the actual communication with the OPB Firmware. An OPB object is given access to an OPB Driver by the Director component (see below). The Director component is the object which creates and ultimately destroys all OPB objects.

The main purpose of the OPB classes is to provide a layer of abstraction between the OPB hardware and the end user. This allows the user to use a generic software mechanism for controlling any type of OPB in any configuration. As a result the user software will be less complex. This abstraction is provided by ensuring that all OPB classes implement a common interface of functions. The user then programmatically uses this interface to control an OPB object. The user does not need to know the type of OPB, or its configuration to control it.

The other great advantage of abstraction is that if a new communication mechanism is developed to talk to an interface board (eg via the USB, PC (PCMCIA), PCI or ISA bus), the OPB class software does not need to be altered in any way to talk to OPBs in the new configuration. Instead it would simply use a new OPB Driver class which encapsulates the necessary knowledge of how to communicate using the new mechanism.

The OPB classes attempt to represent the actual physical interface boards in a software manner. The physical OPBs contain components such as the network transceiver and audio codec ICs, and the micro-controller Firmware. These are components that a D2B Optical developer would want to manipulate and control. The OPB classes also therefore contain

subsidiary software components to represent the corresponding hardware components. These software components are classes in themselves. The present system defines four such classes (described in more detail below):-

- Network Transceiver 816
- Codec 818
- Network Management 820
- Interface Firmware 822

An OPB object 814 creates the subsidiary component objects, depending on the interface board type, and what hardware is on the actual OPB. It then gives users of the OPB object access to these other component object via functions in its API. The user can then use the appropriate component object to the access the functionality of that hardware component on the physical interface board.

In one example, the OPB type of interface board has an audio codec CS4225 IC and network transceiver OCC8001 IC on it. When OPB object 814 is created, it creates a Network Transceiver object 816 and a Codec object 818. The OPB class exposes functions GetComponent("Network Transceiver",0) and GetComponent("Codec",0) in its API to allow access to these objects. The Network Transceiver and Codec classes expose an API to allow the user to control the corresponding hardware on the physical interface board. The index (here '0') indicates the number of the component on the board. This allows for multiple, identical components on one board (for example, with two transceiver ICs, the second is accessed with GetComponent ("Network Transceiver",1).

This approach gives certain advantages :-

- Modular decomposition. The functionality of the OPB class is partitioned into a number of separate software modules. Thus the complexity of the OPB can be broken down into its simpler component parts. This leads to simpler software and easier maintenance.
- Component Re-Use. As the components are in separate modules, they can easily be shared between other software modules. For example, the Network Transceiver and Codec classes are used by both the OPB and AOPB classes, since equivalent ICs are present on both types of board. The code for the components is centralised, so changes to a component need only be made in one place.

- Extensibility. As a consequence of the above points, if a new piece of hardware is added to an OPB (for example. a DSP IC), then one simply develops a new component class for it, and "plugs it in" to the OPB class. This process can be performed with very little change to the OPB class itself.

Each of the component objects created by the OPB must ultimately talk to the interface board firmware to perform the desired actions on the hardware. As all communication with the firmware has to go via the OPB Driver objects, it would seem necessary for each of the components to be able to access the relevant OPB Driver object 812. However, this would tend to negate the advantages of having the OPB classes in the first place (to abstract the communication mechanism with the firmware). Each component would have to implement specific code to talk to every type of OPB in any configuration.

To alleviate this problem, all communication from one of the OPB's component objects 816, 818 to the actual interface board go via the OPB object 814 itself. This means the component objects do not have to embed specific knowledge of the OPBs and communication mechanisms in their code (as this is in the OPB object). It does mean however that the OPB classes must expose strictly defined (but polymorphic) interfaces for these component objects to talk to the firmware. The component objects cannot make any assumptions about which type of OPB class they are connected to, because they can be connected to different types of OPB objects (see Component Re-Use point above). The component objects must however know how to talk to any OPB class type they are connected to. This implies that all OPB class objects must provide the defined interfaces for those components.

The OPB classes in this system therefore expose a number of interfaces for other components to access specific functionality on the interface boards. The OPB classes do in fact implement four different interfaces:-

- Client interface. This is the main interfaces exposed to end user applications via which they can access all the functionality of the OPB (or rather all the functionality the OPB allows the end user to access).
- Register interface. This interface is used specifically by the Network Transceiver and Codec objects to perform register read/writes in the network transceiver IC.

- Transceiver support interface. This interface is used exclusively by the Network Transceiver object 816 for sending and retrieving D2B messages from the OPB object, and setting its D2B address.
- Control interface. This is used exclusively by the Director 824 to initialise and manipulate it the OPB object.

To access some specific functionality of an OPB, the client (user) is forced to go via the appropriate component object that provides it - they cannot go via the client interface. In other words, to perform a register write to the transceiver IC on the OPB, the user must use Network Transceiver object 816. The main OPB client interface does not expose functions to do write to transceiver registers directly. Enforcing this idea ensures that there is only one way to perform a certain unique action. This generally simplifies software and makes maintenance easier.

One other important aspect of the development system is that it notifies end user applications of events occurring on an interface board and in the wider network. These events are passed to OPB Driver object 812 by the OPB firmware, which then passed them to the OPB object 814. OPB object decodes these messages and puts them into a format recognised by the rest of the development system. Only the OPB class understands the meaning of the messages passed by the OPB Driver. It is therefore important that OPB Drivers for a specific interface board type use the same format for these messages. This is important to maintain the abstraction of the OPB class from its OPB Driver classes.

Once the OPB has decoded a message, it passes it to the Network Transceiver object 816, where it will eventually be propagated to an end user application.

Network Transceiver 816

The Network Transceiver class provide an API for control of the actual network transceiver IC on an interface board, and retrieving information regarding the network transceiver. For example there is a Network Transceiver class encapsulating knowledge of the Conan transceiver IC, type OCC8001.

The main aim of the Network Transceiver classes is to provide a layer of abstraction between the end user and the Transceiver IC on the OPB. This idea is almost exactly the same as that of OPB class abstraction detailed in the previous section in that it allows the user to use a generic software mechanism for accessing the functionality of any type of network

transceiver. As a result, the user can write a single piece of software to access the functionality of any Network Transceiver class. This abstraction is provided by ensuring that all Network Transceiver classes implement a common interface of functions. The user then uses this interface by programming to control a Network Transceiver object. The user does not need to have any specific knowledge of a Network Transceiver object to use it.

The Network Transceiver classes provide access to four main areas of functionality.

- Reading and writing to registers.
- Retrieving information about the registers (for example their names etc.)
- Retrieving information about the network transceiver (for example its type, how many registers it has etc.)
- Sending and retrieving D2B messages.

The first three areas of functionality are contained in the main generic interface common to all Network Transceiver classes. This is the API exposed to the client application 810. Another API is provided for the sending/retrieving of D2B messages. This API cannot be used directly by the client, it is only used internally by other development system components (namely the Message Control component).

The reason for this is again related to abstraction. Currently the transceiver IC uses the D2B Optical control frame format. If in future the control frame format were to change to suit a different platform, then its API for sending/retrieving D2B messages would be different of the other Network Transceiver objects. This means that the API could not be abstracted (because the function for sending messages would be different for different network transceivers). The concept of a D2B message could instead be abstracted, and Network Transceiver classes would implement a common API to send them. This would, however, require the Network Transceiver classes to understand the concept of a D2B message, which is judged too high a level concept for the Network Transceiver class. In addition, there is extra management code needed for sending D2B messages, which every user should not be expected to write. Therefore access to the D2B Optical message API is not given to the end user. Instead applications must go via the System Control component API to send and receive D2B messages.

Network Transceiver objects 816 are created, destroyed and owned by an OPB object in the development system environment. The Network Transceiver object maintains a connection to the OPB object's Register and Transceiver support interfaces, via which it accesses the OPB firmware 822. Client 810 is informed of the register that was written to, and the value written to it. OPB object 814 is also able to pass events to the Network Transceiver object. To do this the Network Transceiver classes provide a common generic API for the OPBs to use. This API must be generic (for abstraction) so OPB objects can treat their transceivers transparently (that is, they can inform any type of transceiver of events using the same mechanism/code).

As events from OPB object 814 must ultimately go to the client application 810, Network Transceiver object 816 must somehow propagate the messages to the application. It does this via a Message Control object. The Message Control object is the "bridge" between the components representing the OPB hardware (OPB, Transceiver, Codec components) and the "system instance" components (Message Control, Automatic Response System (ARS), Device components). The "system instance" components are unique to each application using the development system. Each application will have its own unique instances of a Message Control, ARS, Device and System Control components. This is in contrast to the "hardware components" (OPB, Transceiver, Codec etc.) which are shared between development system applications. One instance of each of these hardware objects exists for each corresponding piece of real hardware.

The Message Control object 826 therefore provides the bridge between the hardware component and the development system components. A Network Transceiver object has the ability to connect to a Message Control object, and disconnect from it. Functions to do this are in the same interface on the Network Transceiver object as the D2B message functions. In the process of establishing a "connection", a Network Transceiver object obtains a reference to an interface on the Message Control object which the Transceiver uses to inform the Message Control object of received OPB events and D2B messages. Once the connection is established, a client application can send and receive D2B messages to and from the interface board hardware. The API on the Network Transceiver classes to make the connection must again be generic, so that a Message Control object can connect to any type of Network Transceiver object without having to worry, or even know about the type of Transceiver it is connecting to.

One final service of the Network Transceiver classes is to inform the client application when a register write is performed on the transceiver IC using a development system Network

Transceiver object. When this happens, the Network Transceiver object generates a message and passes it up to the Message Control component 826, and ultimately to the client application. The message contains information indicating which register was written to, and the actual value written to it.

Codec 818

Since the type of network described provides for source data streams in addition to control and data messages, the Codec classes provide an API for control of the audio or video Codec IC on an interface board, and retrieving information regarding the Codec. For example, there is a Codec class encapsulating knowledge of the Codec CS-4225 IC.

As with the Network Transceiver class the main objectives of the Codec classes are to provide a layer of abstraction between development system applications, and different Codec IC types that might be on an OPB. The Codec has the following areas of functionality :-

- Setting/Retrieving Codec properties.
- Reading and writing to registers.
- Retrieving information about the registers (names etc.)
- Retrieving information about the Codec (its type, how many registers it has etc.)

The Codec object 818 in this embodiment has four properties :-

- Mute.
- Volume
- Balance.
- Fade.

These properties are not attributes of the Codec IC itself, but development system imposed representations. It abstracts the meaning of certain Codec IC registers (namely the output attenuation registers), to a level which has meaning to the user. In other words, if the Codec IC's output attenuation registers are all zero, this has no meaning to an end user. If instead we say the Codec IC's *Volume* is 100%, then this does have relevant meaning to the user. Thus *volume* is an abstract representation of the attenuation registers, and it has meaning to the user. These properties can be set and retrieved via the Codec object main client API.

A Codec component is always created by an OPB object and connected to that OPB object in a very similar manner to a Network Transceiver object. The Codec object 818 uses the OPB object's Register Interface to talk to the OPB firmware to perform register writes. Whenever a register write is made to the Codec IC using Codec object 818, the Codec object informs the client applications which registers were written to, and what values were written to them. It does this by calling another function in the OPB object's Register Interface. The Codec object is never notified of events by the OPB object.

Director 824

Director object 824 is in essence an interface board manager component. It is unique in that only one instance of a Director will ever be running/existing at any time. This provides centralised interface board management. The Director exposes a single API which contains function that allow a client to gain access to any OPB object that is running. To be able to do this, the Director knows about all the OPB objects that exist in the environment. It has this knowledge by the virtue of the fact that it is the Director object itself that actually creates (and ultimately destroys) all OPB objects. This does also imply that the Director has specific knowledge of all types of OPB classes and how to create them.

The first thing the Director will do when it is created is to determine what interface boards are attached to the PC. It does this by creating an instance of every type OPB Driver object that it can, and using them to detect the actual OPBs. If an Optical Power Board is detected, the Director then creates an appropriate OPB object, and links that OPB object to the OPB Driver object 812. The OPB object can then freely communicate with the OPB via the OPB Driver. The Director continues to create Driver and OPB objects of each specified type until it can detect no more of them.

For example when the Director is created it creates a Parallel OPB Driver object. It then calls an OPB detection function in the object. The function returns False meaning there is no OPB attached to the PC in parallel, so the Director destroys the Parallel OPB Driver object. It then creates a Serial OPB Driver object and calls its OPB detection function. The function returns true implying the PC has an Optical Power Board connected by serial RS-232. So the Director instantiates an OPB object 814 and passes it a reference to the Serial OPB Driver object. The Director does not proceed to attempt to detect more serial OPBs as it has explict knowledge there will never be more than one. Finally the Director creates an Advanced OPB (AOPB) Driver object and detects an AOPB with it. An AOPB Component object is created

and hooked up to the AOPB Driver object. The Advanced OPB provides the daisy-chain facility explained in the set up of Figure 10. As the Director knows there may be other AOPBs (as in Figure 10, for example) it creates another AOPB Driver component and tries to detect another AOPB. It continues to do this until it has detected all the interface boards attached to the PC.

The Director stores references to all the OPB objects it creates that successfully detected an interface board. It can then provide these references to any requesting client.

The only other area of functionality that the Director is envisaged to have as an option is for navigating Device objects 830 etc. in the same manner as OPB objects. This would allow applications to "spy" on other development systems running (as mentioned in the Device section). To do this the Director object would typically expose another API with functions to allow Device objects to *register* and *un-register* themselves with the Director, and a function for an application to retrieve references to the registered Device objects.

Message Control 826

The Message Control component class provides five main categories of functionality:-

- D2B message queuing and sending
- Receiving and translation of D2B control frames.
- Receiving and propagation of OPB events.
- Long D2B Optical message management.
- Creation and destruction of connections to Transceiver and ARS objects.

The primary function of the Message Control class is to provide D2B message queuing and transmission services to client applications. To be able to transmit a message, a Message Control object must first of all be connected to a Network Transceiver object 816 (and thus indirectly to the physical interface board 804/806 etc). The Message Control class exposes an interface with functions to connect and disconnect it from a Network Transceiver object. The Message Control object is given a reference to a Network Transceiver object's interface, through which it determines the type of Network Transceiver object it is being connected to. The Message Control then gives the Network Transceiver object a reference to one of its interfaces. This interface is the one which the Network Transceiver object subsequently uses to

pass events and D2B messages to the Message Control object. The Message Control object can expose a number of different such "notification" interfaces. This is to provide future extensibility and flexibility as follows.

The novel development system is intended to be extensible and "future proof". One of the values of the Message Control class is that the D2B control frame structure (Figure 7) may change in the future (e.g. maybe for a new type of Transceiver IC), or the same protocols are to be implemented on a different type of network altogether. Therefore the Message Control class should be capable of interacting with different Network Transceiver objects that *may* use different control frame formats. However, the development system components and client applications deal with the concept of D2B application protocol messages, not control frames. A D2B application protocol message contains the real *semantics* of a message (the meaning), whereas the control frame structure is simply a communication *format* imposed on application protocol messages by the transceiver hardware. Development system end users are interested in sending D2B messages, and are oblivious to the actual format via which these messages are sent via the D2B optical network. The D2B message can be thought of as *abstracting* the meaning of the message, from the underlying physical format with which it is communicated.

Client applications and the development system components therefore use the concept of D2B messages, although the same frame format will not apply in application to other types of network or other protocols. The "hardware" components (Transceiver and OPB classes) deal with control frames. Somewhere in the development system framework D2B messages must be translated into control frames and vice versa, and this is the job of the Message Control class. The Message Control object 826 knows about all control frame formats. When the Message Control object 826 is first connected to Network Transceiver object 816, it determines the transceiver type and thus its control frame format. Subsequently when a D2B message is sent, the Message Control can convert the D2B message into the appropriate control frame structure that the Transceiver is expecting. Different Network Transceiver classes will expose different APIs for sending the different control frame types (thus this API cannot be abstracted). This is the reason why end user applications are not in the present embodiment given access to the message sending API of the Network Transceiver classes.

When a control frame is received, Network Transceiver object 816 passes it up to Message Control object 826. As different Transceiver types may use different control frames, the Message Control object provides an API for each control frame type that exists. When the

Message Control is connected to a Network Transceiver object, it gives the Transceiver a reference to the appropriate Message Control notification interface through which the Transceiver can pass its specific control frame types to the Message Control. The end result of this is that the Message Control component can send and receive D2B messages via any type of transceiver IC. Meanwhile, the development system components and end user applications can continue to use the idea of D2B messages without having to alter software in the future to take advantage of new transceiver types.

When application 810 sends a D2B message, Message Control object 826 puts the message in a buffer instead of immediately attempting to transmit it. Periodically the Network Transceiver object 816 to which it is connected signals the Message Control object, indicating that it can go ahead and proceed to send a D2B message if it wishes. Message Control object 826 is then free to pluck a D2B message from its buffer and transmit it. It can then retrieve the transmission result and inform the client application. Placing messages in a buffer allows the development system to perform extra communication management (see for example Long message hereafter), and save applications having to implement buffering themselves.

When Network Transceiver object 816 receives an incoming control frame it passes it on to Message Control object 826. Message Control object 826 converts the control frame into a D2B message and passes that up to client application 810.

Message Control object 826 is therefore responsible for informing development system client applications of three things:-

- Received D2B Optical messages.
- Transmission results for transmitted D2B Optical messages.
- Events received from the underlying development system hardware components.

To achieve this a Message Control object can be connected to an ARS object (828, described below), in the same manner a Network Transceiver is connected to a Message Control object. The functions to do this are even in the same API as used by the Network Transceiver object to connect to the Message Control object.

Message Control object 826 therefore has two APIs :-

- A client/control interface for
  - Connecting the Message Control with a Network Transceiver object.
  - Connecting the Message Control with an ARS object.
  - Sending D2B messages.
- A notification interface used by the connected Transceiver to
  - pass received control frames to the Message Control object.
  - pass hardware component message to the Message Control.
  - inform the Message Control it is free to send a D2B message.

The final service provided by the Message Control class in this example is in relation to long D2B messages. Long D2B messages cannot be transmitted in a single control frame, so extra management code is needed to package long D2B messages into multiple control frames and transmit them. Similar management code is required by the receiver of a long message to accumulate and package control frames into a long D2B message. In addition, certain protocols (for example for aiding flow control) may be required in transmitting and receiving the control frames.

As all the long message logic would have to be implemented by every development system application wishing to deal with long messages, Message Control class implements it instead. It does this internally using a *long message protocol* (LMP) class object. This object contains all the knowledge required for the sending and receiving of long messages. This class can be updated to use any new long message protocol developed in the future. The current implementation contains the logic to package/unpackage control frames into D2B messages (and *vice versa*), and could also be adapted to include flow control protocol logic. The end result will always be that the user is shielded from the long message protocols, enabling them to transmit and receive long messages with relative ease and no extra coding. There can be means to instruct the Message Control object to use a particular long message protocol, or turn off long message processing altogether.

Whenever the Message Control object receives a long message, it passes it to the LMP object for further processing. The LMP object performs any necessary actions building up the complete message from the control frames. When the message is complete it is then passed up to the ARS object 828 by the normal method. When a long message is being transmitted, the LMP object along with the D2B message object itself interact to manage and package the

message into control frames for sending. Once all the control frames have been transmitted, the client application is then informed of the transmission result.

ARS (Automatic Response System) 828

The ARS class provides the foundation for adding intelligent automated response to a development system client application. The class provides the following areas of functionality:

- System and device specification.
- D2B message validation
- D2B message translation.
- Automated device behaviour/response.

One of the driving motivations behind the present development system is the fact that each designer/manufacturer of D2B Optical products will initially develop their own D2B network system. This system will contain specified D2B Optical devices which communicate using a specified subset of the D2B Optical protocols. D2B system integrators would like to be able to test that the devices in the system conform to the protocol and system specification (for example that they do not send D2B messages invalid for the actual configuration of the system). They may also wish to construct device emulators, and test applications to automate the testing of their D2B Optical system or D2B Optical devices/products.

Central to supporting this idea in the development system is the concept of a Protocol Database encapsulated in the System Description File (SDF) 838. The SDF :-

- Contains all the application protocols for a specific D2B Optical version.
- Specifies the subset of those protocols valid for the particular system.
- Specifies the devices that are valid for the system.
- Specifies the valid sub-devices in the devices, and those sub-devices source data types.

Thus an SDF can be created to encapsulate the specification of any specific D2B Optical system, down to which devices (CD changer, telephone and so forth) are actually present within it. The ARS can be instructed to use a specific SDF via a function in an API that it exposes. It can then utilise the knowledge in the SDF, as well as exposing another separate interface via which other development system components can access the same information.

As can be seen in Figure 8, all D2B message communication to and from client application 810 must pass through the ARS object 828. The ARS object can therefore examine every message being sent or received to test whether the message conforms to the protocols and (in this particular system), the system specification described in the SDF. If the client application attempts to transmit an *invalid* message, the ARS can reject the message (not send it), or simply warn the application. If an invalid message is received, the ARS can warn the client application it is invalid. The behaviour of the ARS object 828 in response to invalid messages can be configured via a function in the ARS API. Through the use of the SDF, the ARS can therefore provide an automated D2B message validation service to the client application 810, without the client having to perform any work.

Another feature of the ARS is the symbolic translation of messages. Each D2B application protocol message, for example, are specified as an *Opcode* and a number of *Operands*. This is a very unintuitive representation to an end user, as it is complicated, and is difficult to remember the meaning of a string of digits. To alleviate this, the novel development system can translate D2B messages into a human readable textual representation, and vice versa. This service is provide by the ARS object 828. Thus client application 810 can specify a D2B Optical message to be transmitted in text, and the ARS object will convert the text into the corresponding D2B Optical message. Similarly D2B messages received by the ARS object can be converted into text before being propagated up to the client application. Textual translation in the ARS can be turned on or off by a function in its API.

The ARS object 828, like the Message Control object 826 and Network Transceiver object 816, is connected to other development system components. An ARS object 828 is connected to a Message Control object 826, from which it receives notifications (D2B messages, transmission result and events). After processing, the ARS object propagates all received notifications up to the Device object 830 (see below). The ARS class therefore provides functions to establish connections with the Message Control and Device objects, and ultimately destroy those connections. These functions are provided in a dedicated API.

The ARS class presently provides four APIs for :-

- Setting ARS attributes (eg. which SDF to use, whether to use text translation etc.).
- Connecting the ARS to other development system components.
- Receiving notifications from the connected Message Control object.

- Providing access to the SDF system specification.

Device 830

The Device component can be thought of as a component manager and is the central component in a system instance. The concept of the system instance was described previously but is described in more detail here for clarity.

When a client application 810 uses the development system, a System Control object 832, Device object 830, ARS object 828, Message Control object 826, and (optionally) a Switchbox object 834 are created. These objects (described in more detail below) effectively belong to the application and together form the application's system instance. These objects are unique to the application 810 and cannot be accessed by other applications. This in contrast to the other development system components such as the OPB, Network Transceiver, Codec and Director objects (described below). Those are shared components which can be accessed and used by multiple applications. The client application can control the attributes and behaviour of its development system via a number of APIs.

When a client application uses the development system to control an OPB, it is effectively acting as a D2B device, which has a special status in the network and application protocols. This D2B device has attributes such as a D2B address (distinct from its physical, ring position), whether it is the D2B system master, and so on.. The development system Device component 830 is the object via which the client application can set and retrieve these attributes. In addition, the development system as a whole has attributes, for instance whether its performs textual translation of D2B messages, or uses a Switchbox component. Although the first attribute is effectively an attribute of the ARS 828, and the second is an attribute of the development system as a whole, it is the Device component 830 which *encapsulates* this information about the complete development system being used by the application. It exposes an API through which the various attributes can be set and retrieved.

When a client application uses the development system it creates a System Control object 832. The first thing this object does is to create a Device object. The Device object 830 then proceeds to create the other components of the Development system (namely Message Control, ARS and Switchbox) and connects them together. It then accesses (creates if necessary) the global Director object 824 and retrieves a free OPB object 814. Finally it gets access to the or a Network Transceiver object 816 of the OPB and links its own Message

Control 826 and the Network Transceiver object 816 together. This provides a path from the OPB Driver components 812 all the way up to the Device object 830. When the client application terminates, the Device object unlinks all the components it connected and destroys the objects in its system instance.

The Device object 830 determines initially which OPB object (and hence which physical OPB) a client application's system instance will use. The Device 830 exposes an API through which the client can get access to the OPB object, its Transceiver and Codec objects (see below). However there may be multiple OPBs attached to a PC and the end user may wish to select a particular one to use. The client application can navigate the OPBs attached to a PC via the Director component 824, select one, and instruct its system instance to use it. The Device API provides a function to do this, and the Device object takes care connecting the newly specified hardware components to its own system components.

The Device object like other components must play its part in passing events from the connected hardware components (802, 804) up to the client application 810. It creates and connects to itself an Automatic Response Server (ARS) object 828 thereby allowing the ARS to pass events up to it from the Message Control 826. However the Device object is unique in that it exposes a well defined standard API to allow other components to connect themselves to the Device object for the purpose of receiving notifications from it. As can be seen from Figure 8, the Device is connected to both the System Control object 832 and Switchbox object 834. Once the System Control object has created the Device object it uses the standard API to hook up to it. Similarly once the Device object creates the Switchbox object, the Switchbox object uses the same API to hook itself to the Device object. Subsequently, all notifications received by the Device object are automatically propagated up to the System Control and Switchbox objects.

This standard mechanism could be used by any component to hook up to a Device object. For instance one may construct GUI applications that can "spy" on client applications by hooking up to a specific Device component. There are two criteria for components wishing to do this:-

- The component must implement a specific API through which the Device object will pass the notifications to the component.
- The component must first have some reference to the Device object to connect to it. This implies that the other Development system component that already has a reference

to the Device object supplies that reference, or an external agent (such as the Director component 824) supplies it (this means there would have to be a mechanism via which the Director could be made aware of all running Device objects).

System Control 832

The System Control object 832 is the *interface* for the client onto the Development system environment. Every client application 810 wishing to use the development system must instantiate a System Control object in its application. This object then acts as that application's "bridge" or interface to control the development system. The System Control object 832 exposes a single API to the client application and also expects the application to expose a defined API through which the System Control object can pass notifications to the application.

When the System Control object 832 is created, it instantiates a Device object 830 which then proceeds to construct the rest of the development system specific to the corresponding application 810 (referred to above as the system instance). The System Control object 832 itself contains virtually no functionality, and simply delegates all its API calls to the connected Device object. The System Control object's main job is to translate data passed in the API function calls from the application into the appropriate format used internally by development system, and vice versa.

Once the System Control object 832 creates the Device object 830, it uses the Device's standard mechanism to hook up to it, so the System Control object can receive the notifications from the Device object. The System Control object therefore exposes two interfaces:-

- A client application interface for the application 810 to control the development system.
- A notification interface used by the Device object to notify the System Control object of events.

When client application 810 finally terminates, System Control object 832 severs its connection with the Device object 830, then destroys the Device object. The Device object destroys and "cleans up" the rest of the system instance in the process of its own destruction.

It is possible for a single application to instantiate more than one System Control object 832. In this case, each one created will construct its own entirely unique system instance,

separate from that of the others. Each of the different System Control objects can be used to control a separate interface board (OPB object 814).

Switchbox 834

One of the most important advantages of the D2B Optical technology is the ability to transport multiple channels of source data across a single optical fibre. To control the management of source data transport, D2B source data protocols and D2B Switchbox sub-device have been defined. Every D2B device must implement software to support the Switchbox and its associated source data protocols. This implies that any client application controlling an OPB via the development system, must also implement this software for the "Development system device" (such as the telephone emulation 109' in Figure 9) to inter-operate in a D2B Optical network. This is of course an unwanted burden on all development system applications.

To alleviate this problem the development system has a Switchbox class of object. Switchbox object 834 effectively encapsulates and implements the D2B switchbox and source data connection protocol logic on behalf a client application using the development system. When any D2B message is received, Device object 830 passes it to Switchbox object 834, which then examines the message to see if is a switchbox message. If it is, Switchbox object 834 processes the message and performs any necessary actions, namely propagating new switchbox messages onto the optical network, and controlling source data routing within its own network transceiver.

When a Device object creates a Switchbox object, it gives the Switchbox access to the Network Transceiver object 816 being used by the system instance. The Switchbox object can then freely alter the transceiver routing information table (RIT) or equivalent routing control, in response to D2B switchbox messages. If the client application 810 instructs its system instance to use a different OPB, Device object 830 will ensure that the Switchbox object is always provided with access to the correct Network Transceiver object to manipulate.

As the RIT of different transceiver ICs may be different (number of ports, port size, organisation), and the Switchbox could be connected to any of the different transceiver types, the Switchbox object encapsulates specific knowledge for each transceiver type. Ideally the Switchbox should contain only generic behaviour, and each Network Transceiver class should expose an API that allows the Switchbox to access specific knowledge regarding that

Transceivers RIT. The Switchbox object would then use this knowledge dynamically to perform the correct RIT operations for that transceiver. However, the current Transceiver and Switchbox designs do not support this idea.

Client application 810 may wish to route source data on or off an optical network to emulate a real device, as shown in Figure 4. This source data would be routed via a specific port on the transceiver IC, and would logically be associated with a D2B sub-device. Switchbox object 834 cannot determine this information itself, and is therefore informed of it by the client application. To this end, the Switchbox object exposes an interface via which it can be *configured*. The user can access this API via Device object 830. This API allows the user to specify which D2B sub-devices are logically connected to the transceiver IC, as well as the actual input/output port on the transceiver to which they are connected. The Switchbox object uses the ARS object to access the SDF system specification to determine the source data type associated with a particular sub-device. Thus when the Switchbox object needs to route source data from/to a sub-device it knows which RIT channel to manipulate, as it has knowledge of the transceiver port, and the format of the data associated with the port.

When the Switchbox is involved in source data connection building, it informs the client application by sending it notifications. These notifications include information such as whether a connection is being built/destroyed, the ID for the connection, and the sub-device(s) affected by the connection. In addition, channel information associated with the connection (that is, which channels within the source data fields (Figure 5) the connection is using) is also given. Switchbox object 834 passes these notifications to the Device object 830 to which it is connected. Device object 830 already exposes an API to the ARS object 828 for receiving notifications. The Switchbox object uses the very same API.

The Switchbox itself exposes three interfaces :-

- A client interface for *configuring* the Switchbox.
- A control interface used by the Device object to initialise and connect the Switchbox object to a Network Transceiver object.
- A notification interface via which the Device object passes notifications to the Switchbox.

Up to this point it has been assumed that client application 810 will *always* want to use the services of the Switchbox component. However this may not be the case, as the user may

want to implement their own Switchbox logic and operation themselves. To support this, Device object 830 in its main API provides a function to toggle the use of the Switchbox component. If this attribute is off, the Device object does not even create the Switchbox object (or destroys if it does exist). It will subsequently create the Switchbox object if the attribute is toggled on. If toggled off the client application is totally responsible for implementing switchbox behaviour in their application, by manipulation of transceiver registers and appropriate handlign of D2B messages.

Device Database 836

A Device database 836 can also be provided, in which the protocols and features actually implemented by a range of specific products are encoded in a pre-arranged format. This "library" of device descriptions can be used together with the ARS object 828 to emulate a selected product within the development system, without the user programming an application specifically to emulate it. Manufacturers can share and expand this library of device descriptions as new products are developed, to facilitate greatly the development of compatible products.

Client Events

The system instance is not just a passive system which responds to commands from a client application. It can inform the client of events that happen at any time, asynchronously, via the System Control object 832. These events can originate from things occurring on the external D2B Optical network, in the Interface firmware 822, and in the system instance itself. In the present example, the events in the system instance can be generalised into the following categories:-

- Received D2B messages
- D2B message transmission results for messages sent by a client application.
- Network Management state changes and the causes for the changes. Many different events lie in this category, as many different things will cause the state changes eg. Lock lost/gained, timeouts, network errors etc.
- Network transceiver register writes. These occur when any network transceiver register is successfully written to.
- Internal Errors. For example if a client application tries to set some system instance property to an invalid value, or an error occurs in the internal system framework.

The following table defines some of the events that the System is capable of signalling to a client application.

Event Type	Description
Message Received	Fired when client receives an application protocol message. The message is passed with the event.
Optical Wake-up	Fired when the optical wake up trigger on the OPB is triggered.
Electrical Wake-up	Fired when the electrical wake up trigger on the OPB is triggered.
Remote Start-up	Fired once the optical ring is operational after start-up
Remote Shutdown	Fired once the optical ring has been shutdown
Lock Lost	Fired when client's network transceiver loses lock of the network bit stream
Lock Gained	Fired when client's network transceiver gains.
Fault Report	Fired when a network management fault is reported eg. Device lost.
Nodes in Ring	Fired each time the Master detects a new device in the ring after startup. The number of nodes is passed in the event.
Startup Failure	Fired when the network fails to startup correctly (see PCT/GB98/ (62796WO).
Bi-Phase/Parity Error	Fired when network transceiver detect a bi-phase or parity error in the network bit stream.
OPB comms error	Fired when communication is lost between an OPB Driver and the interface board on which a client is controlling a network transceiver.
Node Position	Fired once a network transceiver has established its node position after start up. The node position is passed in the event (see EP-A-0725516 (95P03)).
Message Transmitted OK	Fired after each successful transmission of an application protocol message on the network. The actual message, or an identifier is passed with the event so the client can see which message the event concerns.

Message Failed.	As Message Transmitted OK but fired when a message fails to be transmitted.
RIT changed.	Fired when a switchbox alters a network transceiver RIT for source data routing purposes. The RIT register and its new value are passed in the event. The connection ID associated with the event could also be passed.
Address Initialised	Fired once a network transceiver has established its device address after a successful network startup. The address is passed in the event (see PCT/GB98/00872 (62792WO).
In alarm mode.	Fired when Firmware goes into alarm mode.
In fault recovery mode.	Fired when OPB Firmware enters fault recovery mode.
Light Received.	Fired when network transceiver receives light after the electrical wakeup is triggered.
Lock Timeout	Fired if a device does not gain lock within a set period from the electrical wakeup being triggered
Report Position Timeout	Fired if a device does not receive a Report Position status request within a set period after gaining lock at start up.
Shutdown Timeout	Fired when a Master device does not receive a Power Off status report from all devices within a set period after shutdown

Development System Initialisation Example

Figure 11 illustrates the sequence of action and communication between various component modules of the development system in initialisation. The actions of each component are represented on a respective time line, and labelled with the same reference signs as in Figure 8.

At step 1100, the client application 810 in order to begin operation, creates an instance of System Control object 832. At 1102, the System Control object in turn creates the Device object 830, which itself at 1104 creates the director object 824.

At 1106, Director object 824 creates an OPB object 814. The OPB object at 1108 to 1114 creates the Network Transceiver object 816, the Codec object 818, and initialises the Firmware 820 and Network Management 822 elements of the interface board.

At 1116 and 1118, the Device object 830 further creates the ARS object 828 and Message Control 826 respectively. At 1120, the ARS object 828 establishes link with the corresponding Message Control object 826.

At 1124 the Device object 830 sends the command Get Component ("OPB", "0") to the Director 824 which at 1126 returns the identity of the OPB 814 reserved to this application. At 1128 the Device object 830 requests OPB 814 to confirm that it is free and confirmation is received in step 1130. At 1132, Device object 830 informs System Control 832 that this instance of the development system is ready for operation.

Development System Operation Example

Figure 12 illustrates the sending of a D2B message on to the network. At 1200 the client application makes a function call to system control 838, providing the D2B message that it wishes to send in a textual representation, such as "<PLAY><FORWARD>", addressed to the "CD CHANGER". At 1202, System Control 838 passes this call to the device object 830, which in turn at 1204 passes it to the ARS object 828. The ARS object 828 checks the message according to the warning level which has been set, and according to protocols and system description in the system description file 838. The message is either accepted or rejected.

In the case when the error is rejected, at 1206 ARS 828 sends an error code to device object 830 which at 1208 passes the same error code onto the system control object. At 1210, an event is signalled to the client application 810, which includes the relevant error code.

Assuming that the message passes the test according to the warning level and the system description file, the ARS 828 translates the text format message into hexadecimal, and passes it at 1212 to message control 826. The hexadecimal format for the above message example would be 0x1c8, 0x190, 0x0e, 0x42, 0x90, 0xc3, 0x75, where the first two codes are the source and destination addresses. At 1214, message control object 826 passes the message

to the network transceiver object 816, which in turn passes it at 1216 to the OPB object 814. OPB object 814 now sends the message using the OPB driver, into the interface board and onto the D2B Optical network.

At 1218, the transmission result is passed by the Network Transceiver object back to Message Control object 826. From there it is passed at steps 1220 to 1224 via the ARS 828 and Device object 830 to System Control 838. Finally, at 1226 System Control object 832 signals an event to the client application 810, including transmission information, and a representation of the complete message.

The person skilled in the art will readily appreciate that many modifications of the above embodiments are possible within the spirit and scope of the invention. Furthermore, the development system described and any of its key features, may be adapted for use in protocols other than the D2B Optical system, and for transceiver components other than the Conan product range.

CLAIMS

1. A system comprising computer implemented components to be used in conjunction with a hardware network interface connected to a functioning communication network for the emulation of a specific product under development, said components providing an interface between a client program and the network interface for the transmission of messages to other devices on the network, and incorporating a protocol database, wherein at least one of said components is arranged for processing messages instructed by the client program with reference to the protocol database, so as to detect illegal messages being instructed for transmission onto the network.

2. A system as claimed in claim 1, wherein application-level protocols specific to the functionality of devices in the network are defined and checked by means of said protocol database.

3. A system as claimed in claim 2, wherein different application-level protocols specific to the different functionality of different devices in the network are defined and checked by means of said protocol database for conflict with the specific type of device being emulated.

4. A system as claimed in claim 3, further comprising a system description specifying the type or functionality of other devices present in the network, whether as finished products or by emulation, wherein at least on of said components is arranged for processing messages instructed by the client with reference to said system description so as to detect inappropriate messages being instructed for transmission onto the network.

5. A system as claimed in any preceding claim, wherein the protocols define at least one sequence of behaviour, where extended co-operation between plural stations in the network is necessary to achieve a desired result.

6. A system as claimed in claim 5, wherein said sequences include at least one sequence for network management, independent of specific device functionality.

7. A system as claimed in claim 5, or 6, wherein said sequences include at least one sequence of behaviour for establishing the routing of data streams carried via the network in addition to the said messages.

8. A system as claimed in claim 5, 6 or 7, comprising a specific component for automatic implementation of said sequence of behaviours, without involvement of the client program.

9. A system as claimed in claim 8, wherein all messages received from the network are passed to the or each specific component, and further passed to the client program if not relevant to the sequence of behaviour implemented by the specific component.

10. A system as claimed in any preceding claim, wherein said components take the form of programming objects, which exchange messages with one another according to predetermined rules.

11. A system as claimed in any preceding claim, wherein certain components may be shared between different applications active in the same computer hardware, while others are reserved to a single active application.

12. A system as claimed in claim 11, wherein those components which are shared may include components which provide the interface to specific network interface hardware.

13. A system as claimed in any preceding claim wherein an action level may be set to determine whether detected illegal messages are to be transmitted or blocked.

14. A system as claimed in any preceding claim wherein a reporting level may be set to determine whether detected illegal messages are to be reported, for example reported to the client program.

FIG. 1

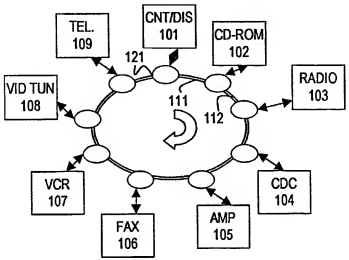

TEL.
109
CNT/DIS
101
CD-ROM
102
121
RADIO
103
VID TUN
108
111
112
VCR
107
CDC
104
FAX
106
AMP
105

FIG. 2

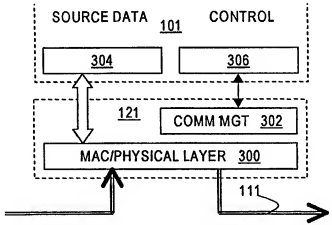

SOURCE DATA
101
CONTROL
304
306
121
COMM MGT 302
MAC/PHYSICAL LAYER 300
111

FIG. 3

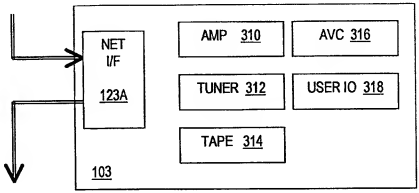

NET
I/F
123A
AMP 310
AVC 316
TUNER 312
USER IO 318
TAPE 314
103

FIG. 4

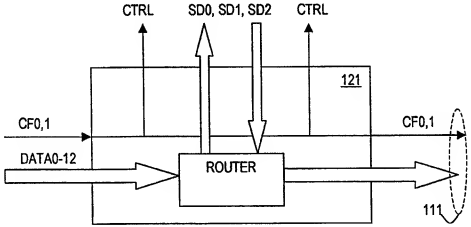

CTRL
SD0, SD1, SD2
CTRL
121
CF0,1
CF0,1
DATA0-12
ROUTER
111

FIG. 5

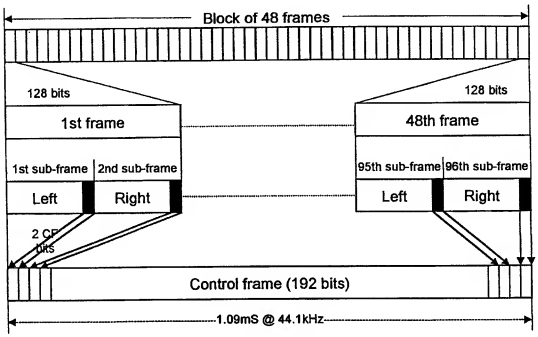


FIG. 6

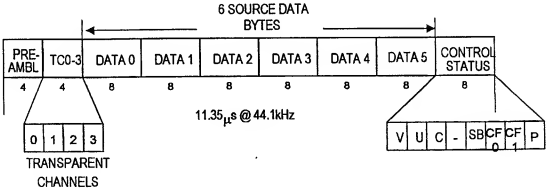


FIG. 7

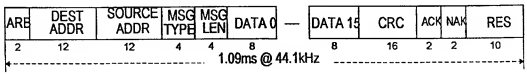


FIG. 8

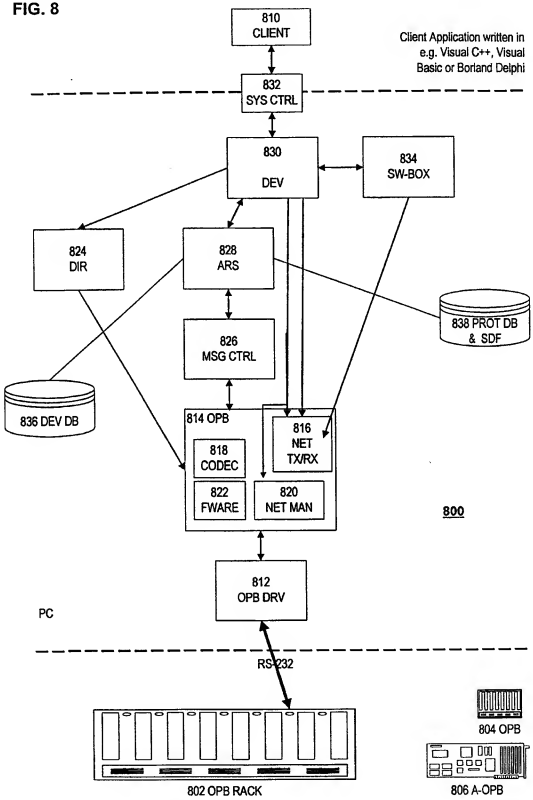

810
CLIENT
Client Application written in
e.g. Visual C++, Visual
Basic or Borland Delphi
832
SYS CTRL
830
DEV
834
SW-BOX
824
DIR
828
ARS
838 PROT DB
& SDF
826
MSG CTRL
836 DEV DB
814 OPB
816
NET
TX/RX
818
CODEC
822
FWARE
820
NET MAN
800
812
OPB DRV
PC
RS-232
804 OPB
802 OPB RACK
806 A-OPB

FIG. 9

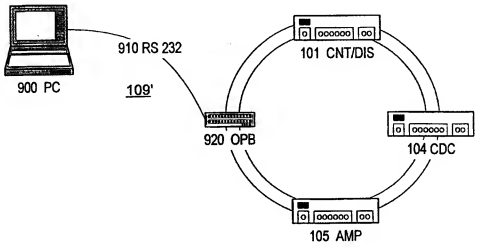

910 RS 232
900 PC
109'
101 CNT/DIS
920 OPB
104 CDC
105 AMP

FIG. 10

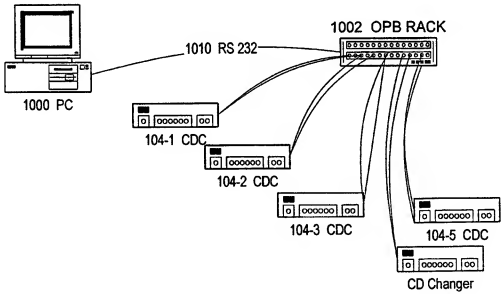

1002 OPB RACK
1010 RS 232
1000 PC
104-1 CDC
104-2 CDC
104-3 CDC
104-5 CDC
CD Changer

FIG. 11

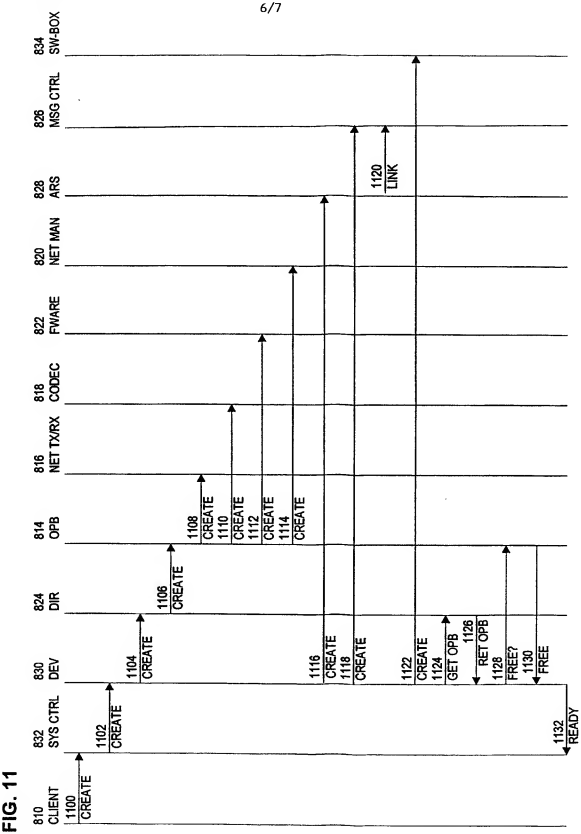
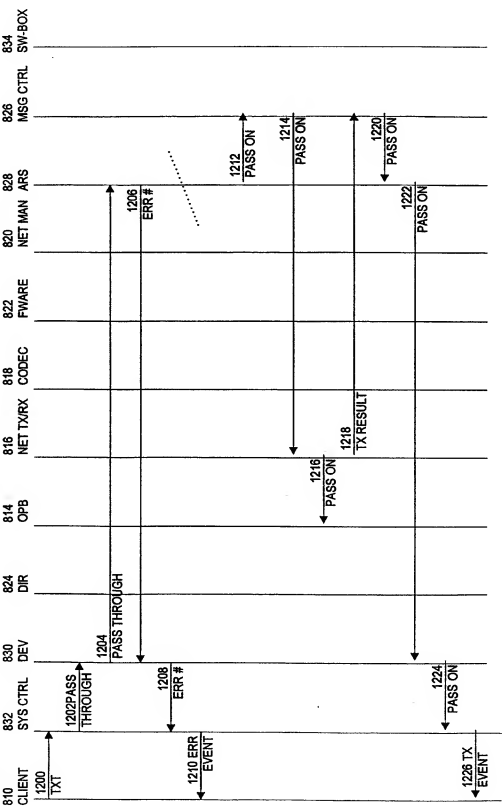

810 CLIENT
832 SYS CTRL
830 DEV
824 DIR
814 OPB
816 NET TX/RX
818 CODEC
822 FWARE
820 NET MAN
828 ARS
826 MSG CTRL
834 SW-BOX
1100 CREATE
1102 CREATE
1104 CREATE
1106 CREATE
1108 CREATE
1110 CREATE
1112 CREATE
1114 CREATE
1116 CREATE
1118 CREATE
1120 LINK
1122 CREATE
1124 GET OPB
1126 RET OPB
1128 FREE?
1130 FREE
1132 READY

FIG.12


810 CLIENT
832 SYS CTRL
830 DEV
824 DIR
814 OPB
816 NET TX/RX
818 CODEC
822 FWARE
820 NET MAN
828 ARS
826 MSG CTRL
834 SW-BOX
1200 TXT
1202 PASS THROUGH
1204 PASS THROUGH
1206 ERR #
1208 ERR #
1210 ERR EVENT
1212 PASS ON
1214 PASS ON
1216 PASS ON
1218 TX RESULT
1220 PASS ON
1222 PASS ON
1224 PASS ON
1226 TX EVENT